분산 환경 시스템에서 RMI를 이용한 원격 메소드 연결에 관한 연구

소경영*, 최유순*, 박종구**

A Study on the Remote Method Connection using RMI in the Distributed Computing System

Kyung-Young So, Yue-Soon Choi, Jong-Goo Park

요 약

Java RMI는 네트워크를 통해 연결된 분산 환경 시스템에서 원격 컴퓨터에 존재하는 메소드간에 정보를 주고받을 수 있는 어플리케이션을 개발할 수 있도록 개발된 모델이다. 본 논문에서는 Java RMI를 이용하여 원격 컴퓨터에 존재하는 객체의 메소드를 연결하는 시스템을 설계하고 구현하였다. 이를 위해, 클라이언트가 이용할 수 있는 동적 메소드 연결 인터페이스 및 구현 부분을 API 형식으로 완성하였다. 또한 동적 메소드 연결시에 서버에서 사용하는 동적 메모리 할당, 소멸 등을 위한 관리 루틴을 완성하였다.

ABSTRACT

In this paper, we design and implement of the remote method connection system using Java RMI in the distributed computing system. In pursuing this goal, we implement the dynamic method connection interface and API. And then we describe the dynamic memory management routine.

키워드

컴포넌트, 연결자, RMI, 가상연결

* 원광대학교 컴퓨터공학과 박사과정

** 원광대학교 컴퓨터 및 정보통신공학부 교수

1. 서 론

최근 들어 컴퓨터 하드웨어 기술의 급속적인 발전 및 이를 지원할 수 있는 소프트웨어 기술이 발전됨에 따라 분산 시스템 환경을 지원할 수 있는 다양한 연구가 진행되고 있다. 분산 시스템은 컴퓨터 네트워크를 통해 연결된 호스트 컴퓨터들이 모여서 이루어져 있고 분산 시스템 소프트웨어라는 것이 설치되어 있어서 컴퓨터들의 동작을 조정하고 시스템 자원이 공유되는 특징을 가지고 있다[2].

분산 시스템의 어플리케이션은 분산 컴퓨팅 방식과 병렬 컴퓨팅 방식으로 구분된다. 분산 컴퓨팅 환경에서는 네트워크로 연결된 컴퓨터들이 하나의 작업을 분산 작업하는 방식으로서 클라이언트/서버 모델 방식과 객체 기반 모델 방식으로 구분된다. 클라이언트/서버 모델 방식은 분산 시스템에서 가장 기본적인 방식으로서 클라이언트 프로세서와 서버 프로세서로 구성된다. 서버 프로세서는 자원을 관리하는 관리자의 역할을 수행하며 클라이언트 프로세서는 공유된 하드웨어와 소프트웨어 자원을 접근하여 이용한다. 이러한 클라이언트/서버 모델 방식에서 수행되는 처리 방식은 클라이언트 프로세서와 서버 프로세스간에 통신을 수행하여 데이터를 주고받은 방식으로 이루어지는데 이 때 클라이언트와 서버가 통신할 때 클라이언트와 서버가 동시에 이용할 수 있는 통신 규칙인 프로토콜을 이용한다. 객체 기반 모델 방식에서는 서비스를 요청하는 클라이언트와 서비스를 제공하는 서버가 잘 정의된 캡슐화 인터페이스를 사용하여 분리한 것이다. 즉, 클라이언트가 데이터와 실행 코드로 구성된 서비스 구현 부분으로부터 분리되어 구성된다. 따라서 객체 기반 모델 방식에서 클라이언트가 객체에게 메시지를 보내면 이 객체는 메시지를 읽어 제공해야할 서비스의 종류를 결정한다[2].

분산 환경 시스템 환경에서 어플리케이션을 구성하기 위해 사용되는 클라이언트/서버 모델 방식에서 이를 구현하기 위해 자주 사용되는 방식으로 소켓 방식이나 원격 프로시쥬어 호출 방식을 이용한다. 소켓을 사용하여 클라이언트/서버 모델 방식을 지원한다는 것은 클라이언트와 서버가 데이터를 주고받는 데 필요한 모든 명령어 방식과 환경을 지원하는 프로토콜까지 지원한다는 의미를 갖는다. 하지만 원격 프로시쥬어 호출 방식은 클라이언트와 서버 사이의 인터페이스를 로컬 프로시쥬어 호출 방식의 개념을 추상화시킨 고수준의 방식이다[13].

분산 네트워크 환경 및 이기종 기계 환경에서 응용 소프트웨어 개발을 위해 개발된 Java 언어는 객체지향 특성을 지원하는 언어이며 Java 프로그래밍 언어 환경에서는 이식성, 번역성, 고성능, 및 단순성 등을 지원하고 있다. 이러한 Java 언어를 이기종 기계 환경 및 분산 네트워크 환경에서 응용 소프트웨어를 원활히 수행하기 자바 가상 기계를 이용하며 자바 가상 기계는 Java 인터프리터를 이용하여 프로그램을 실행한다. 특히, Java 언어는 RMI 방식 등을 이용하여 네트워크 및 분산 환경에 적합한 환경을 지원하고 있다. RMI는 네트워크를 통해 다른 호스트 컴퓨터상의 Java 객체의 메소드를 호출할 수 있는 어플리케이션을 개발할 수 있도록 준비된 모델이다. 이 모델에서는 데이터를 주고받는 분산 객체가 모두 Java 언어로 구현되어야 하는 특징을 가지고 있다. 따라서 RMI는 다른 언어로 구현된 객체간의 메소드 호출은 지원하지 않는다[4][5].

본 논문에서는 이러한 분산 시스템 환경에서 Java 언어로 구현된 객체의 메소드간에 데이터를 주고받을 수 있는 기법을 RMI를 활용하여 더욱더 개선하고자 한다. 이를 위해 그림 1과 같이 구성된 모델을 제시하고 이를 지원할 수 있는 환경을 구현하였다.

원격 메소드 호출을 통해 데이터 전달한 후 결과 값

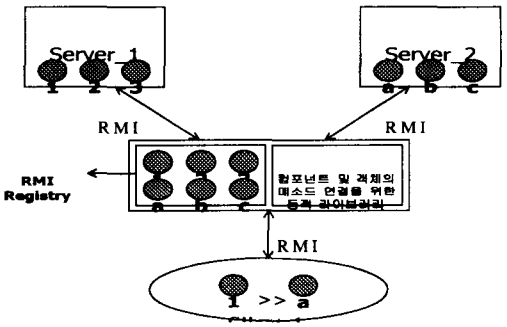

그림 1. 시스템 구성 모델

을 전달받기 원하는 클라이언트(Client_1)는 먼저 원격 컴퓨터(Server_1)의 상의 객체 메소드인 ①을 호출한다. 기본적으로 서버 컴퓨터상의 메소드를 호출하기 위해서는 RMI 레지스트리를 통해 이루어진다.

Server_1의 결과 값을 다시 Server_2상의 메소드에 전달하기 위해서는 기존 처리 방식에서는 결과 값을 클라이언트가 결과 값을 돌려 받은 후 이를 저장하여 다시 Server_2에 존재하는 원격 메소드 호출 시에 매개변수의 형태로 전달해야 하거나 별도 방식을 이용해야 한다. 하지만 본 논문에서는 Server_1의 결과 값을 곧바로 Server_2의 원격 메소드에 직접 전달할 수 있는 동적 라이브러리를 구축하였다. 따라서 Server_2상에서는 클라이언트로부터 전달되는 원격 메소드의 호출에 필요한 매개 변수 및 Server_1에서 수행된 결과 값을 직접 전달받을 수 있는 효과를 가진다. 또한 Server_2에서는 클라이언트의 요구에 대한 메소들 호출 수행 결과를 클라이언트로부터 전달된 Server_1의 결과 값에 직접 반영할 수 있는 효과를 가진다. 이러한 원격 메소드간에 정보를 주고받을 수 있는 환경 구축을 위해 기존의 RMI에 동적 라이브러리를 구축하였으며 Server_1으로부터 전달되는 결과 값을 저장하고 관리하기 위한 메모리 할당, 관리, 소멸 동작을 위한 루틴을 구현하였다.

본 논문의 구성은 제 2장에서 Java 언어를 이용하여 구현된 객체의 원격 메소드 호출에 활용되는 RMI에 대해 자세히 고찰한다. 제 3장에서는 RMI를 활용하여 원격 객체의 메소드 연결을 위한 시스템 구성과 구현 원리 및 활용 예제에 대해 자세히 설명한다. 마지막으로 제 4장에서는 결론과 본 연구에서 지속적으로 수행해야될 향후 연구 방향에 대해 기술한다.

2. Java RMI를 활용한 분산 컴퓨팅

2.1 Java RMI

Java Remote Method Invocation기술을 JDK 1.1부터 적용되기 시작했으며 Java 환경에서 원격 컴퓨터 또는 프로그램간에 통신을 할 수 있는 기능을 제공한다. RMI 통신의 특징은 객체사이에서 이루어지는 객체 지향적 통신 방법이며 간단하면서도 강력하다는 것이다[4][5]. 이와 유사한 CORBA, DCOM, EJB 등과는 달리 객체 또는 컴포넌트간 정보 전달을 지원하는 미들웨어 시스템을 별도로 설치할 필요도 없으며 자바 가상 기계 자체에 이미 RMI 환경이 구현되어 있다. 따라서 분산 환경에서 간단한 객체 통신의 방법으로 시스템을 구현하기 위해 Java RMI가 널리 활용되고

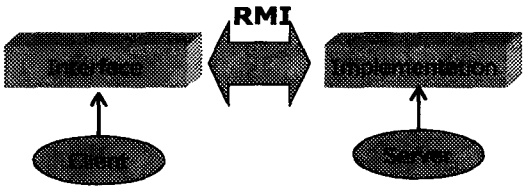

그림 2. Java RMI 통신

있으며 RMI는 Java 프로그램 사이의 통신이 그림 2와 같이 클라이언트/서버 구조를 가지고 있다.

기본적으로 클라이언트와 서버는 동일한 인터페이스를 가지고 있으며 서버는 이러한 인터페이스를 실질적으로 구현하고 있다. 클라이언트는 RMI를 이용하여 서버에 존재하는 메소드의 구현 부분에 서비스를 요청하며 서버는 요청된 클라이언트의 요구에 대한 수행 결과를 RMI를 이용하여 반환한다.

2.2 RMI의 구조

RMI 시스템은 크게 그림 3과 같이 스터브/스켈레톤 계층, 원격 참조자 계층, 트랜스포트 3 계층이 독립적으로 구성되어 있다.

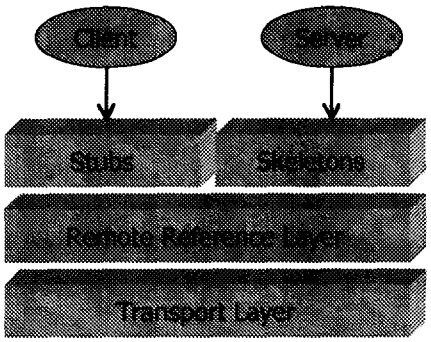

그림 3. RMI 시스템 구조

각각의 계층은 인터페이스로 구성되어 있으며 각각의 프로토콜로 정의되어 있다. 스터브/스켈레톤 계층은 사용자 어플리케이션과 시스템 사이의 인터페이스 역할을 한다. 따라서 어플리케이션을 개발할 때 클라이언트 부분과 서버 부분에서 각각 생성되며 마샬링/언마샬링을 통해서 클라이언트와 서버사이에서 전달된다 즉, 스터브는 원격 메소드 호출을 개시하고 서버로 전달할 매개 변수를 마샬링하며 원격 참조자 계층

에게 원격 메소들 호출이 발생되었음을 통보한다. 또한 서버에서 반환되는 원격 메소드 호출의 반환 값을 다시 언마샬링 하여 클라이언트에 전달한 후 원격 참조자에게 원격 메소드 호출이 종료되었음을 알린다. 스켈레톤은 클라이언트에서 전달한 매개 변수를 언마샬링하고 실질적인 메소드 호출에 대한 수행을 진행한 후 결과 값을 다시 마샬링하여 클라이언트에게 전달한다. 원격 참조자 계층은 원격 참조자나 호출 프로트콜을 지원한다. 트랜스 포트 계층은 서로 다른 주소 공간 사이에서 마샬링된 스트림을 전달하는 하위 수준의 계층이다. 이 계층은 주소 공간에 대한 연결 상태를 설정하고 관리하며 원격 객체에 대한 정보를 저장할 수 있는 테이블을 관리한다.

실질적으로 RMI를 이용한 객체의 메소드간 데이터 전달을 위한 구조는 원격 메소들 호출을 원활히 진행하기 위한 다음과 같은 기능을 수행한다.

인터페이스 정의 부분 : 원격 컴퓨터에 존재하는 메소드간에 데이터를 주고받기 위한 통신을 위해 송신하는 클라이언트측과 수신하는 서버 측이 서로 메시지의 형식을 공유해야 한다. RMI 환경에서도 클라이언트/서버간에 통신할 객체의 메소드는 미리 인터페이스로 정의하고 서로 공유해야 한다. 이러한 인터페이스는 RMI 레지스트리에 등록되며 클라이언트는 등록된 메소드의 인터페이슬 참조하여 서버 객체의 메소드를 호출하게 된다.

스터브 : 스터브는 RMI 시스템에서 클라이언트 응용 프로그램이 원격 컴퓨터에 존재하는 메소드를 호출할 때 초기화 및 매개 변수 등을 하나의 단위로 묶는 마샬링 기능한다. RMI 시스템을 통하여 원격 객체를 획득하는 방법은 원격 메소드를 호출하여 반환 값을 객체로 받아들이는 방법이다. 이러한 방법을 응용 프로그램으로 구현하면 매우 복잡하기 때문에 RMI 시스템은 각 원격 객체에 대한 스터브를 생성하여 클라이언트 측에서 사용하도록 한다. 하지만 응용 프로그램 작성자가 스터브를 생성할 필요는 없고 원격에서 사용할 "rmic"를 사용하여 스터브 클래스를 생성한다.

스켈레톤 : 스터브 클래스가 클라이언트 측에서 원격 호출을 조절한다면 스켈레톤 클래스는 서버 측에서 원격 메소드의 매개 변수를 받아들이고 실질적으로 제공해야할 서비스를 조절한다. 스켈레톤도 응용 프로그램 작성자가 생성할 필요가 없이 "rmic"를 이용하여 자동적으로 생성한다. 따라서 스터브와 스켈레톤이 각각 클라이언트와 서버 응용 프로그램에 대한 스터브 클래스 및 스켈레톤 클래스를 생성해야 정상적인 RMI 통신이 가능하다.

클라이언트 응용 프로그램 : 클라이언트 응용 프로그램은 원격 객체의 메소드를 서버로부터 접근 권한을 획득하여 실질적으로 사용하는 메소드를 의미한다. 원격 객체에 대한 메소드를 사용하기 위해 원격 객체에 대한 메소들 사용 권한을 획득하려면 객체의 메소드가 존재하는 서버의 위치를 알아야 한다. 이러한 정보는 RMI 레지스트리가 설치된 컴퓨터에 질의를 한다.

서버 응용 프로그램 : 서버 응용 프로그램은 원격지에서 클라이언트 호출에 대해 서비스를 제공하는 메소드를 의미한다. 즉, RMI 레지스트리에 등록된 메소드에 대한 실질적인 구현 부분을 가지고 있다.

RMI 레지스트리 : RMI 레지스트리는 RMI 통신이 가능하도록 RMI 환경에서 제공되는 기본 이름 서비스(naming service)를 지원하는 프로그램이다. 서버 응용 프로그램은 RMI 레지스트리에 자신을 등록해야만 서비스가 가능하고 클라이언트 응용 프로그램은 RMI 레지스트리에서 서비스를 제공받고자 하는 서버의 주소 및 메소드를 찾는다.

3. 원격 메소드 연결 시스템

3.1 원격 메소드 연결 시스템 모델

본 논문에서는 RMI를 이용한 원격 메소드간의 정보 전달을 보다 효과적으로 수행할 수 있는 시스템을 그림 4와 같이 구성한다.

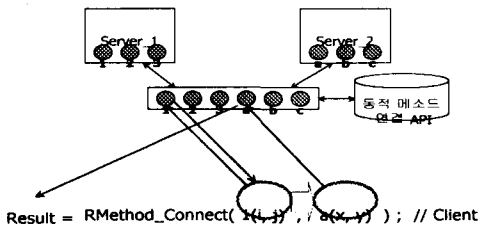

그림 4. 동적 메소드 연결 시스템 모델

클라이언트는 Server_1에 대한 호출을 일반적인 형

태로 수행하여 생성된 결과 값을 다시 Server_2에 전달하는 과정을 본 연구에서 구현한 동적 메소드 연결을 위한 API를 이용한다. 즉, Server_1에서 수행된 결과 값을 Server_2에 전달시에 새로운 공간을 생성하여 전달하며 동시에 Server_2에 존재하는 원격 메소드 호출을 위한 정보도 동시에 전달한다. 따라서 클라이언트는 Server_1의 결과 값을 별도로 Server_2에 전달해야 하는 부담을 줄일 수 있다. Server_2에서는 원격 메소드 호출에 대한 정보 전달 및 Server_1의 결과 값을 임시로 저장할 수 있는 기억 공간을 프로그램 수행 시에 사용할 수 있도록 동적 메모리 공간을 할당한다. 동적 메모리 공간을 관리하기 위해 본 연구에서 구현한 동적 메모리 생성, 관리, 소멸 루틴을 이용한다. Server_2에서는 원격 메소드 수행 시에 생성되는 결과 값을 동적 메모리에 저장된 Server_1의 결과 값에 반영하여 새로운 결과 값을 얻을 수 있다. 동적 메소드 연결 API에서는 Server_1으로부터 전달되는 결과 값에 대한 정보인 결과 값의 자료형, 크기 등을 추출하여 Server_2에 전달 시에 반영하여 동적 메모리 공간에 대한 할당 정보 등으로 활용한다. 이러한 동적 메소드 연결을 위한 인터페이스는 그림 5와 같은 형태를 갖는다.

먼저 동적 연결 메소드 연결을 위해 클라이언트는 RMethod_Connect() API를 이용하여 Server_1에 존재하는 1,(i, j) 메소드를 호출한다. Server_1에서는 원격 메소드에 대한 매개 변수 등을 입력으로 받아 결과 값

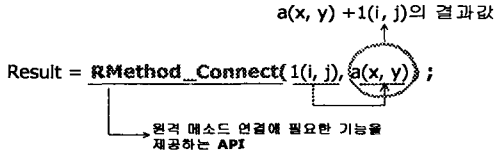

그림 5. 동적 메소드 연결 인터페이스

을 반환한다. 반환되는 결과 값은 또 다른 원격 컴퓨터인 Server_2에 매개 변수 형식으로 전달되며 동시에 실질적인 Server_2에 대한 메소드 호출인 a(x, y)와 동시에 전달된다. a(x, y)에 대한 호출은 RMI를 통해 일반적인 형태로 이루어지며 특별히 Server_1에 대한 결과 값을 전달하기 위해 동적 연결 메소드 API에서 지원하는 라이브러리를 이용한다. 최종적인 Server_2의 결과 값은 결과 값을 전달받고자 하는 "Result" 변수에 저장된다.

3.2 동적 메모리 관리

동적 연결 메소드 API는 원격 서버로부터 전달되는 결과 값을 프로그램 실행 시간에 이용할 수 있도록 저장하고 관리하는 동적 메모리 관리 루틴을 제공한다. 원격 컴퓨터는 전달받은 Java 언어에 대한 수행을 위해 자바 가상 기계에 설치되어 있으며 자바 가상 기계는 Java 스택을 이용하여 기억 공간을 할당하며 특별히 원격 컴퓨터에서 전달되는 결과 값을 저장하기 위해 힙 메모리 공간을 할당한다.

동적 메모리 할당을 위해 실행 시간에 필요한 만큼 원격 컴퓨터로부터 할당받는다. 할당받은 기억 공간은 자유 블록의 리스트 형태로 유지된다. 힙 공간은 헤드와 실제 자료가 저장되는 기억 공간으로 구성되며 헤드에는 힙의 크기와 다음 힙을 가리키는 포인트 정보가 저장된다. 자유 블록은 다음 공간을 가리키는 포인터, 크기, 기억 공간으로 구성되며 블록의 시작에 위치한 제어 정보를 헤더라고 부르며 그림 6과 같은 구조를 가지고 있다. 또한 각 블록의 크기는 정렬을 간소화하기 위하여 헤더 크기의 배수로 이루어진다.

```
typedef union header {
   struct {
     union header *ptr;
     unsigned size;
   } s;
   double x;
} Header;
```

그림 6. 헤더의 구조

힙 공간 할당 함수인 HeapAlloc()이 호출되면 요구한 기억 장소를 우선 자유 리스트 내에 충분한 크기의 블록이 있는지를 최초 적합(first-fit) 방식 또는 최적 적합(best-fit) 방식으로 검사한다. 이때 요구한 크기와 발견된 블록의 크기가 정확히 일치하지 않으면 발견된 블록은 사용자에게 복귀하지만 블록이 너무 큰 경우 필요한 만큼만 사용자에게 복귀되고 나머지는 자유 리스트 내에 남아있게 된다. HeapAlloc() 함수의 실질적인 구현은 알고리즘 1과 같다.

```
void *HeapAlloc(unsigned nbytes) {
   Header *p, *prevp;
```

```
    unsigned nunits;
    if (nbytes > HEAP_SIZE)
        return NULL;
    nunits=

(nbytes+sizeof(Header)-1)/sizeof(Header)+1;
    if( (prevp = __shFreep) == NULL ) {
      ZeroMemory(__SharedHeap,
HEAP_SIZE);
      __shFreep=(Header *)__SharedHeap;

__shFreep->s.size=(HEAP_SIZE)/sizeof(Header);
      __shFreep->s.ptr=prevp = __shFreep;
    }
    for( p = revp->s.ptr; ; prevp = p, p =
      p->s.ptr ) {
      if( p->s.size >= nunits ){
        if( p->s.size == nunits ){
          prevp->s.ptr = p->s.ptr;
        } else {
          p->s.size -= nunits;
          p += p->s.size;
          p->s.size = nunits;
        }
        __shFreep = prevp;
        return (void *)(p+1);
      }
      if (p == __shFreep) return NULL;
    }
  }
```

[알고리즘 1] 힙 공간 할당 알고리즘

더 이상 필요하지 않은 힙 공간은 다시 사용될 수 있도록 우선 자유 리스트 내의 삽입될 적절한 위치의 발견한 후 자유 리스트에 반환한다. 자유 리스트로 반환된 블록이 다른 사용되지 않은 블록과 인접해 있다면 하나의 큰 블록으로 구성된다. 블록이 인접해 있는가의 결정은 자유 리스트 내의 자료 블록들은 주소가 증가되는 순서로 유지한다. 사용된 힙 공간의 반환을 위한 실질적인 알고리즘은 알고리즘 2와 같다.

```
  void HeapFree(void *ap) {
    Header *bp, *p;
    bp = (Header *) ap-1;
    if          (bp->s.size
HEAP_SIZE/sizeof(Header))
      return;
    ZeroMemory(ap, (bp->s.size -1)
          * sizeof(Header));
    for( p = __shFreep;
        !(bp > p && bp < p->s.ptr);
        p = p->s.ptr )
      if( p >= p->s.ptr &&
          (bp > p || bp < p->s.ptr))
        break;
      if( bp + bp->s.size == p->s.ptr ){
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
      } else
        bp->s.ptr = p->s.ptr;
      if( p + p->s.size == bp ) {
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
      } else
        p->s.ptr = bp;
      __shFreep = p;
  }
```

[알고리즘 2] 힙 공간 반환 알고리즘

3.3 동적 연결 메소드의 예

본 연구에서 제시한 동적 연결 메소드 시스템을 이용한 현재까지 실질적으로 구현된 결과를 이용하여 다음과 같은 결과를 얻을 수 있다. 먼저 Server_1에서는 "Hello World!" 출력하는 메소드를 수행하며 Server_2에서는 바탕색을 파란색으로 출력하는 메소드를 클라이언트가 본 연구에서 제시한 원격 메소드 연결 API를 이용하는 경우이다. 먼저 각 서버에서 제공하는 원격 인터페이스는 그림 7과 같다.

```
  //Server_1
  import  java.rmi.*;
  public  interface  Hello_Blue  extends
Remote
  {
```

```
        public  String  sayHello( )
                throws  RemoteException;
  }

  //Server_2
  import  java.rmi.*;
  public    interface    Hello_Blue    extends
Remote
  {
        public  String  bluePainting( )
                  throws  RemoteException;

  }
```

그림 7. 원격 인터페이스

또한 이러한 원격 인터페이스를 위한 각각의 실질적인 구현은 그림 8과 같다.

```
  // Server1 : HelloImpl . java
  import  java.rmi.*;
  import  java.rmi.server.*;
  import  java.net.*;
  public  class  HelloImpl
        extends  UnicastRemoteObject
        implements  Hello
  {
        public HelloImpl( )
                throws  RemoteException  {
             super( );
        }
        public  String  sayHello( )
                throws  RemoteException  {
                  return   Hello,  World! ;
        }
   ...

  // Server2 : bluePaintingImpl . java
  import  java.rmi.*;
  import  java.rmi.server.*;
  import  java.net.*;

  public  class  bluePaingingImpl
        extends  UnicastRemoteObject
        implements  bluePainting
  {
        public  bluePaintingImpl( )
                throws  RemoteException  {
                 super( );
        }
        public  String  bluePaing( )
                throws  RemoteException  {
        return blue(int x, int y, int x) ;
        }
   ...
```

그림 8. 원격 인터페이스의 구현

Server_1에서는 "Hello World!" 문자열을 출력하는 기능을 수행하며 Server_2에서는 파란색을 출력하는 기능을 수행한다. 이러한 메소드의 구현을 이용하기 위해 그림 9와 같이 작성 클라이언트 프로그램을 수행한다.

```
  // Hello_bluePaintingInvoker . java
  import  java.rmi.*;
  import  java.rmconnect.* ;
  public  class  Hello_bluePaintingInvoker
  {
        public  static  main( String  args [ ] )
{
        RMISecurityManager     sm  =  new
RMISecurityMaanger( );
        System.setSecurityManager( sm );
        try  {
        Hello   h  =  ( Hello )Naming.lookup(
 rmi//host/hello  );
             bluePainting  bp  =  ( bluePainting
)Naming.lookup(

 rmi//host/bluePainting  );
                          String       s     =
Rmethod_Connect(h.sayHello( ), bp.bluePaint()) ;
             System.out.println( s );
        }
        catch( Exception  e )  {
```

```
                System.out.println( Exception
:  + e );
        }
       }
  }
```

그림 9. 클라이언트 프로그램

클라이언트 프로그램에서는 원격 메소드 연결 API를 이용하기 위해 "import java.rmconnect.* ;" 문장을 삽입한다. "rmconnect"에는 원격 컴퓨터에 존재하는 메소드의 연결 및 실행을 위한 모든 라이브러리 기능을 포함하고 있다. 실질적인 Rmethod_Connect() API는 Server_1에 존재하는 "h.sayHello()"를 호출하며 반환되는 결과 값을 Server_2에 존재하는 bp.bluePaint() 호출 시에 동시에 전달한다. Server_2에서는 전달되는 결과 값을 힙 메모리 공간을 할당하여 저장하며 bluePaint() 메소드 수행하여 최종 결과를 Heap 메모리에 저장된 "Hello World!"에 반영한다. 따라서 클라이언트에 반환되는 최종 결과 값은 파란색 글자색을 갖는 Hello World!가 출력된다.

4. 결 론

최근 들어 컴퓨터 하드웨어 기술의 급속적인 발전 및 이를 지원할 수 있는 소프트웨어 기술이 발전됨에 따라 분산 시스템 환경을 지원할 수 있는 다양한 연구가 진행되고 있다. 분산 시스템은 컴퓨터 네트워크를 통해 연결된 호스트 컴퓨터들이 모여서 이루어져 있고 분산 시스템 소프트웨어라는 것이 설치되어 있어서 컴퓨터들의 동작을 조정하고 시스템 자원이 공유되는 특징을 가지고 있다.

분산 네트워크 환경 및 이기종 기계 환경에서 응용 소프트웨어 개발을 위해 개발된 Java 언어는 객체지향 특성을 지원하는 언어이며 Java 프로그래밍 언어 환경에서는 이식성, 번역성, 고성능, 및 단순성 등을 지원하고 있다. 이러한 Java 언어를 이기종 기계 환경 및 분산 네트워크 환경에서 응용 소프트웨어를 원활히 수행하기 자바 가상 기계를 이용하며 자바 가상 기계는 Java 인터프리터를 이용하여 프로그램을 실행한다. 특히, Java 언어는 RMI 방식 등을 이용하여 네트워크 및 분산 환경에 적합한 환경을 지원하고 있다. RMI는 네트워크를 통해 다른 호스트 컴퓨터상의 Java 객체의 메소드를 호출할 수 있는 어플리케이션을 개발할 수 있도록 준비된 모델이다. 이 모델에서는 데이터를 주고받는 분산 객체가 모두 Java 언어로 구현되어야 하는 특징을 가지고 있다. 따라서 RMI는 다른 언어로 구현된 객체간의 메소드 호출은 지원하지 않는다.

본 논문에서는 분산 시스템 환경에서 Java 언어로 구현된 객체의 메소드간에 데이터를 주고받을 수 있는 기법을 RMI를 활용하여 더욱더 개선하고자 한다. 클라이언트는 임의의 Server_1에 대한 호출을 일반적인 형태로 수행하여 생성된 결과 값을 다시 Server_2에 전달하는 과정을 본 연구에서 구현한 동적 메소드 연결을 위한 API를 이용하였다. 즉, Server_1에서 수행된 결과 값을 Server_2에 전달 시에 새로운 공간을 생성하여 전달하며 동시에 Server_2에 존재하는 원격 메소드 호출을 위한 정보도 동시에 전달한다. 따라서 클라이언트는 Server_1의 결과 값을 별도로 Server_2에 전달해야 하는 부담을 줄일 수 있다. Server_2에서는 원격 메소드 호출에 대한 정보 전달 및 Server_1의 결과 값을 임시로 저장할 수 있는 기억 공간을 프로그램 수행시에 사용할 수 있도록 동적 메모리 공간을 할당한다. 동적 메모리 공간을 관리하기 위해 본 연구에서 구현한 동적 메모리 생성, 관리, 소멸 루틴을 이용한다. Server_2에서는 원격 메소드 수행시에 생성되는 결과 값을 동적 메모리에 저장된 Server_1의 결과 값에 반영하여 새로운결과 값을 얻을 수 있다. 동적 메소드 연결 API에서는 Server_1으로부터 전달되는 결과 값에 대한 정보인 결과 값의 자료형, 크기 등을 추출하여 Server_2에 전달시에 반영하여 동적 메모리 공간에 대한 할당 정보 등으로 활용한다.

본 연구에서 구현한 동적 메소드 연결 시스템은 클라이언트가 임의의 서버에서 수행된 결과 값을 또 다른 서버에 전달하는 경우에 간편하게 이용될 수 있다. 특히, 임의의 서버에서 수행된 결과 값을 활용하는 경우에 직접 서버로부터 결과 값을 전달받을 수 있으므로 클라이언트는 단지 결과 값을 전달할 수 있도록 본 연구에서 구현한 API를 이용하기만 하면 된다. 본 연구에서 구현된 결과는 현재 다양한 자료형에 대해 수행될 수 있도록 구현하였으며 특히, Java 언어에서 제공하는 기본 자료형에 대해서는 구현을 완료하였다.

향후에 클라이언트의 부담을 줄이기 위해 원격 메소드 간 연결을 전담할 수 있는 에이전트 서버를 이용할 수 있도록 확장할 것이며 이러한 원격 메소드 연결 기능을 다양한 분산 시스템 환경에서 검사한 후 연결의 정확도 및 수행 시간을 평가할 예정이다.

참고 문헌

[1] R. Monson, Enterproce Java Bean, O'Reilly, 2000

[2] Kurt Wallnau, Nelsin Weiderman, Distrubuted Object Technology With CORBA and Java : Key Concepts and Implications, Technical report CMU/SEI-97-TR-004, June, 1997.

[3] Robert Orfall, Dan Harkey, Jeri Edwards, The Essential Distributed Objects Survival Guide, John Wiley & Sons Inc., 1996.

[4] Mary Campione, Kathy Walrath, The Java Tutorial : Object-oriented Programming, Addison Wesly, 1996.

[5] Ken Arnoldm James Gosling, The Java Programming Language, Addison Wesly, 1999.

[6] Robert Orfall, Dan Harkey, Client/Server Programming with JAVA and CORBA, Guide, John Wiley & Sons Inc.,1997.

[7] David Flanagan, Java in a Nutshell, O'Reilly & Associates, Inc., 1996.

[8] Michael Mattsson, "Object-Oriented Frame works", Lund University, 1996.

[9] Digre T., "Business Object Component Archi tecture", IEEE Software pp.60-69, 1998.

[10] Peter Harzum, Oliver Sims, Business Comp onent Factory: A Comprehensive Overview of Component Based Development for the Enterprise, OMG Press, 2000

[11] Kevin J. Sullivan, Mark Marchukov, and John Socha, "Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Modeling", "IEEE Tranactions on Software Engineering, Vol. 25, No.4, July/August, 1999

[12] E. Roman, Mastering Enterprise JavaBeans & the Java 2 Platform, Enterprice Ed, Jone Wiley&Sons. Inc., 1999.

[13] Qusay H. mahmoud, Distributed Programming with Java, Manning, 2000.

박 종 구(Jong-Goo Park)

1969년 동국대학교 농업경제학과 학사

1975년 동국대학교 전자정보처리학과 석사

1999년 동국대학교 통계학과 박사

1981~현재 원광대학교 컴퓨터공학과 교수

※관심분야 : 전문가 시스템, 소프트웨어 공학, 소프트웨어 신뢰성 공학, 모바일 프로그래밍 등

최 유 순(Yue-Soon Choi)

1986년 원광대학교 컴퓨터공학과 학사

1990년 원광대학교 컴퓨터공학과 석사

2000년 원광대학교 컴퓨터공학과 박사수료

※관심분야 : 소프트웨어 공학, 컴포넌트 소프트웨어, xml 등

소 경 영(Kyung-Young So)

1986년 원광대학교 전자공학과 학사

1990년 원광대학교 컴퓨터공학과 석사

1999년 원광대학교 컴퓨터공학과 박사수료

1991년~현재 익산대학 컴퓨터과학과 부교수

※관심분야 : 소프트웨어 공학, xml, 전문가 시스템